Memory allocation method in stack
- Comparison between Intel and GCC compilers

Abstract

This paper explores the calling convention of popular compilers by analyzing the simple source codes of demonstration and provides simple and efficient solutions to iron issues out when SSEx instruction set is adopted in applications. It is very worth referring to its thought for migrating Unix/Linux applications to Microsoft Windows platform.

Introduction

The trends of software industry are apt to port Unix/Linux applications to Microsoft Windows platform. Four factors have been combined to speed alternative up.

● The increasing performance and low power consumption in Intel CPUs. If compared with the traditional RISC(Reduced Instruction Set Computing) CPUs, Intel CPUs have obtained the leading position by making full use of the multi-core, hyper-thread, media tech(AVX/SSEx/MMX) and turbo boost, etc.

● The power of individual IA(Intel Architecture) CPUs has continued to grow, following and even outpacing the Moor’s law. The volume of several PCs can replace a traditional RISC server, and the dominance will be enlarged unceasingly in future.

● Windows7 OS issued optimizes the high techs of Intel CPUs, such as multi-core, hyper-thread, etc, so the performance of applications is dramatically improved too. In addition, the friendly UI attracts users for Microsoft Windows Platform.

● The rich resources of Unix/Linux application owning stable and high efficient attributes are valuable for reuse in the development of applications. The cost saving pushes trends to be accelerated as well.

The rest sections are organized as below.

● Section 2 introduces what compilers are used.

● Section 3 explores memory allocation method in stack, and it contains two sub-sections for GCC compiler and Intel compiler respectively.

● Section 4 shows solutions basing on the result of section 3, and lists their strength and weakness.

● Section 5 summarizes the whole paper and draws the conclusion.

Compilers

Intel provides the powerful C/C++ compiler to build applications with ease, but there is not,
unfortunately, a cross-compiler for Microsoft Windows Platform. Unix/Linux applications heavily depend on the syntax of GCC, for an instance, GCC uses AT&T inline ASM and ICL(Intel Windows C/C++ compiler) uses Intel inline ASM, so ICL fails to analyze them in the source codes. Moreover, the different object formats(ELF for Unix/Linux and PE/COFF for Microsoft Windows) lead to the errors of linker. As a result, GCC cross-compiler of Microsoft Windows must be adopted to integrate two parts of source codes. Figure 1 shows a redeveloping application basing on Unix/Linux applications.

![Figure 1 Redeveloping applications in Microsoft Windows](image)

Hybrid development gracefully solve these building issues, however, it is a huge challenge of compatibility among compilers. To some extended, especially for applications encapsulated by SSEEx instructions, they may crash randomly. The unpredictability of applications extremely raises the cost of maintenance. The section 3 demonstrates different memory allocation methods in stack which is a key point during the migration.

**Memory allocation method in stack**

Stack is an essential element in applications, and its variable memory allocation methods among compilers bring about some conflicts even collapse in runtime. By investigating these methods, the ideal solution will be found. The next two sub-sections seek for both GCC and Intel compilers’ methods by linking the source codes of demonstration.

**GCC memory allocation method in stack**

Instance is a good tutor on expounding problems. The section complies with the rule to clearly explain its memory allocation method in stack. Figure 2 prints the whole source codes of foo_gcc which executes SSEEx instructions marked by red color inside. In fact, this red block is meaningless in the programming logic, but it is useful to lay the memory of stack out.
The source codes of ASM embody the whole information of layout, so Figure 3 lists its contents for reference.

In accordance to its working order of Figure 3, the flow is:
1. Save ebp and esp in the first place.
2. Allocate stack’s space for the buf variable. To align 16bytes for its address, high 8bytes are discarded and assign low 16bytes to the buf variable.
3. Execute the movdqa instruction of SSEx.
4. Restore stack and return the result.

Figure 4 discloses the layout of stack after the analysis. Some conclusions are drawn in light of this clue:
- The address alignment of stack is decided in the compiling time, and it is a kind of static allocation method. eip and ebp inappropriate 4bytes respectively, so the offset of the buf variable is 24bytes.
- The address of calling point must be aligned by 16bytes, at least it is a rule when SSEx is adopted.
- The extension of stack from high address to low address, and eax carries the returning value.
Figure 4 memory layout of foo_gcc in stack

**Intel memory allocation method in stack**

To format a coherent style, this sub-section follows the former, and the related results are made on the basis of the analysis. Figure 5 prints the whole source codes of foo_icl invoking foo_gcc after an SSEx instruction. As mentioned before, this SSEx instruction is only to make the layout of stack better.
Figure 5 foo_icl C source codes

The source codes of ASM embody the whole information of layout, so Figure 6 lists its contents for reference.
; -- Machine type FU
; mack_description "Intel(R) C++ Compiler Professional for applications
running on IA-32, Version 11.0  Build 20091105 48"
; mack_description "-w"
; .64SF
; 387
; OPTION DOTNAME
; ASSUME CS:FLAT,DS:FLAT,ES:FLAT
; TEXT SEGMENT PARA PUBLIC FLAT 'CODE'
; CONDAT _foo_icl

TEXT:?
; -- Begin _foo_icl
; mack_begin
IF @Version GE 800
; .MMX
ELSEIF @Version GE 612
; .MMX
; HHWORD TEXTEQU <QUARD>
ENDIF
IF @Version GE 800
; .XMM
ELSEIF @Version GE 614
; .XMM
; HHWORD TEXTEQU <QUARD>
ENDIF
ALIGN 16
PUBLIC _foo_icl
_foo_icl PROC NEAR
; parameter l: i64 + esp
.B1.1:
    push ebp
    mov esp, ebp
    add esp, -16
    push esi
    sub esp, 10
; LOE ebx edx
    ; Freqs .B1.0

.B1.2:
; Begin ASM
    lsa esi, DWORD PTR [esp]
    mov edx HHWORD PTR [esi], xmm0
; End ASM
    ; Freqs .B1.1

.B1.3:
    push DWORD PTR [8+ebp]
    call _foo_gcc
    ; LOE eax ebx edx
    ; Freqs .B1.2

.B1.4:
    imul eax, eax
    add esp, 16
    pop esi
    mov esp, ebp
    pop ebp
    ret
; 15.15
; 15.15
; 15.15
; 15.15

ALIGN 16
; LOE
; mack_end:
_foo_icl ENDP
_foo_icl ENDS

_DATA SEGMENT DWORD PUBLIC FLAT 'DATA'
_DATA ENDS
; -- End _foo_icl
_DATA SEGMENT DWORD PUBLIC FLAT 'DATA'
_DATA ENDS
EXTERN _foo_gcc;PROC
; INCLUDELIB <libnat>
; INCLUDELIB <libcmp>
; INCLUDELIB <libsrc>
; INCLUDELIB <sval_dsrc>
; INCLUDELIB <oldlibem>
; INCLUDELIB <libdecimal>
END
In according to its working order of Figure 6, the flow is:
1. Save ebp and esp in the first place.
3. Allocate stack’s space for the buf variable. To align 16bytes for its address, high 12bytes are discarded and assign low 16bytes to the buf variable.
4. Execute the movdqa instruction of SSEx.
5. Call foo_gcc function.
6. Restore stack and return the result.

Figure 7 discloses the layout of stack after the analysis. Some conclusions are drawn in light of this clue:
- The address alignment of stack is decided in the runtime, and it is a kind of dynamic allocation method. ‘and esp, 16’ ensures the correction of address in the demonstration.
- Compiler doesn’t guarantee the 16bytes’ alignment in the calling point. It is 4bytes’ alignment in the demonstration.
- The extension of stack from high address to low address, and eax carries the returning value.
Figure 7 memory layout of foo_icl in stack
Solution

GCC requests that the address of calling point must be 16bytes' alignment, and Figure 4 lays its stack out. If caller violates this calling convention, the failing static allocation method causes the illegal execution of SSEx, and applications crash in the runtime. The allocation method of ICL(referring to Figure 7), unfortunately, hits this point because it doesn’t guarantee the essential rule of GCC. This section will discuss dual avenues to solve the gap as follows.

Use compiling options

The high version of GCC compiler, for example v4.3.0, has added an new compiling option named *mstackrealign* which generates an alternate prologue/epilogue that realigns the runtime stack at entry on the Intel X86. Figure 8 prints the ASM source codes after adding *mstackrealign* to build applications.

```assembly
.file "foo_gcc.c"
.text
.globl _foo_gcc
.def _foo_gcc; .scl 2; .type 32; .endef
_foo_gcc:
  leal 4(%esp), %ecx
  andl $-16, %esp
  pushl -4(%ecx)
  pushl %ebp
  movl %esp, %ebp
  pushl %ecx
  subl $20, %esp

/App
  # 7 "./foo_gcc.c" 1
  movdqa %xmm0, -24(%ebp)

  # 0 "" 2
/W0_APP
  movl (%ecx), %eax
  jnull (%ecx), %eax
  addl $20, %esp
  popl %ecx
  leave
  leal -4(%ecx), %esp
  ret
```

Figure 8 foo_gcc ASM source codes realigned stack
The original 7 instructions are expanded to 15 instructions if compared with Figure 3. The additional instructions focus on the adjustment of stack’s address to properly execute movdqa instruction.

Use wrapper functions

Wrapper functions play a same role when paralleling with compiling options in dealing with the issues of layout. The appropriate allocation of stack is designed manually to match the special
calling convention of GCC, and completely solve the conflict.

```assembly
SECTION .text align = 16
extern _foo_gcc

global _foo_gcc_wrapper
align 16
_foo_gcc_wrapper:
push ebp
  mov ebp, esp
  and esp, -16
  sub esp, 12
  mov eax, [ebp + 8]
push eax
call _foo_gcc

  mov esp, ebp
  pop ebp
ret
```

Figure 9 foo_gcc_wrapper ASM source codes

Figure 9 manually allocates the space of stack and guarantees 16bytes’ alignment at the calling point, and its working flow is:
1. Save ebp and esp in the first place.
2. Align esp by ‘and esp, -16’ and invoke foo_gcc after pushing i parameter.
3. Restore stack and return result.

The layout of stack is disclosed by the Figure 10.
Figure 10 memory layout of foo_gcc_wrapper in stack

Wrapper functions are manual mechanisms for the layout of stack in comparison with compiling options. It has its advantages and disadvantages, and table 1 lists these traits.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiling options</td>
<td>✓ Simple compiling configuration ✓ Needn’t change source codes</td>
<td>✓ Low performance ✓ Big binary size ✓ Need high version of compiler ✓ Need whole source codes</td>
</tr>
<tr>
<td>Wrapper functions</td>
<td>✓ High performance ✓ Small binary size ✓ Needn’t source codes</td>
<td>✓ Need high skills of ASM ✓ Need additional source codes for wrapper</td>
</tr>
</tbody>
</table>
Table 1 Comparison between compiling options and wrapper functions

$mstackrealign$ aligns all entries of functions even if they are not necessary at all, so the extra workloads take negative influences of performance and increase binary size generated by compiler, too. The shortcoming can be effectively controlled by handwriting wrapper functions in terms of implementation, but the advanced programming skills are essential to obtain right ASM source codes. Furthermore, none is an ideal solution in all cases, and the better one should balance each other by combination, rather than sacrifice other side.

**Conclusion**

The paper seeks for the layout of stack in linkage with the source codes of demonstration, and describes the mechanism of allocation among compilers. By further analyzing their gaps, two solutions are designed on root causes. Comprehensive approach combined both sides can improve the performance of application and decrease binary size as well, even if source codes are not available. As a result, developers may borrow ideas from the two solutions when migrating Unix/Linux applications to Microsoft Windows platform.